# Syntaxation

# JavaScript:

## The Universal Virtual Machine

# JSLint.

The Good Parts.

`http://www.JSLint.com/`

# WARNING!

JSLint will hurt
your feelings.

Syntax is the least important aspect of programming language design.

Syntax is the least important aspect of programming language design.

Fashion is the least important aspect of clothing design.

# Programming Languages:

## An Interpreter-Based Approach

Samuel N. Kamin [1990]

Lisp

APL

Scheme

SASL

Clu

Smalltalk

Prolog

# Minimal Syntax

Lisp

```
(fname arg1 arg2)
```

# Smalltalk 80

```
object name1: arg name2: arg2

object operator arg

[ var | block ]
```

# IF Statement

And yet don't look too good,
nor talk too wise

```
C       FORTRAN

        IF(A-B)20,20,10
   10   A=B
   20   CONTINUE
```

```
C       FORTRAN IV

        IF(A.LE.B)GO TO 30
        A=B
   30 CONTINUE
```

```
comment ALGOL 60;


if a>b then begin
    a:=b
end;
```

```
// BCPL

IF A > B {
    A := B
}
```

```
/* B */

if (a > b) {
    a = b;
}
```

```
-- Ada

if a > b then
    a := b;
end if;
```

```
¢ Algol 68 ¢

if a > b then
    a := b
fi
```

# Emotional Style

## Fashionable Tolerance of Syntaxtic Ambiguity

a ♣ b ♥ c

((a ♣ b) ♥ c)

(a ♣ (b ♥ c))

# Binding Power

| | |
|---|---|
| 10 | `=` `+=` `-=` |
| 20 | `?` |
| 40 | `&&` `\|\|` |
| 50 | `===` `<` `>` `<=` `>=` `!==` |
| 60 | `+` `-` |
| 70 | `*` `/` |
| 80 | unary |
| 90 | `.` `[` `(` |

# `word`

variable?

statement keyword?

operator?

special form?

( )

Function definition and invocation

Grouping

Separation

# Parsing

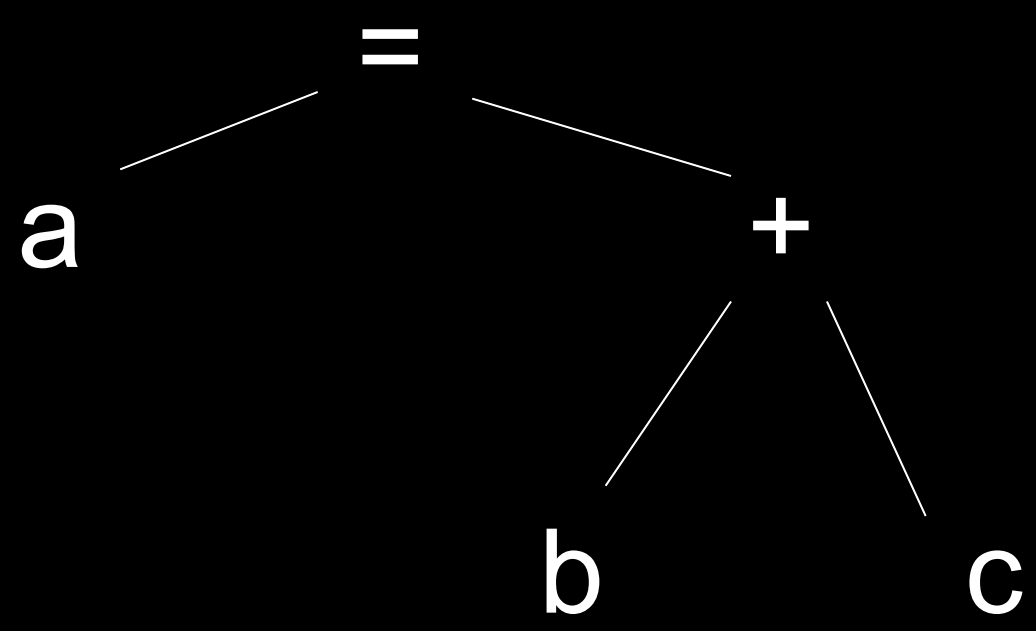
Theory of Formal Languages

Tokens are objects

prototype ← symbol ← token

```
advance()
advance(id)
```

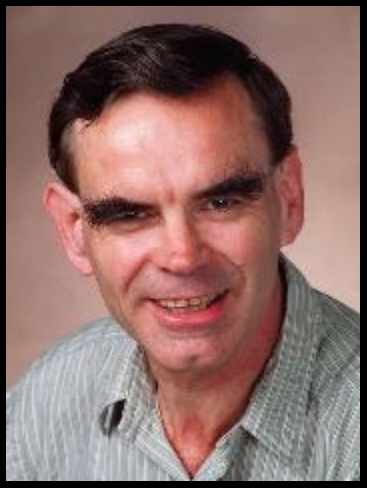
a = b + c;




Weave a stream of tokens into a tree

# Top Down Operator Precedence

- Vaughan Pratt [POPL 1973]
- simple to understand
- trivial to implement
- easy to use
- extremely efficient
- very flexible
- beautiful



# Why have you never heard of this?

- Preoccupation with BNF grammars and their various offspring, along with their related automata and theorems.
- Requires a functional programming language.
- LISP community did not want syntax.
- JavaScript is a functional language with a community that likes syntax.

What do we expect to see to the left of the token?

| | |
|---|---|
| left denotation | led |
| null denotation | nud |

- only nud `! ~ typeof {` prefix
- only led `* . = ===` infix, suffix
- nud & led `+ - ( [`

```
var prototype_token = {
    nud: function () {
        this.error("Undefined.");
    },
    led: function (left) {
        this.error("Missing operator.");
    },
    error: function(message) {
        ...
    },
    lbp: 0      // left binding power
};
```

```
var symbol_table = {};

function symbol(id, bp) {
    var s = symbol_table[id];
    bp = bp || 0;
    if (s) {
        if (bp >= s.lbp) {
            s.lbp = bp;
        }
    } else {
        s = Object.create(prototype_token);
        s.id = s.value = id;
        s.lbp = bp;
        symbol_table[id] = s;
    }
    return s;
}
```

```
symbol(":");
symbol(";");
symbol(",");
symbol(")");
symbol("]");
symbol("}");
symbol("else");

symbol("(end)");
symbol("(word)");
```

```
symbol("+", 60).led = function (left) {
    this.first = left;
    this.second = expression(60);
    this.arity = "binary";
    return this;
};
```

```
symbol("*", 70).led = function (left) {
    this.first = left;
    this.second = expression(70);
    this.arity = "binary";
    return this;
};
```

```
function infix(id, bp, led) {
    var s = symbol(id, bp);
    s.led = led || function (left) {
        this.first = left;
        this.second = expression(bp);
        this.arity = "binary";
        return this;
    };
    return s;
}
```

```
infix("+", 60);
infix("-", 60);
infix("*", 70);
infix("/", 70);
infix("===", 50);
infix("!==", 50);
infix("<", 50);
infix("<=", 50);
infix(">", 50);
infix(">=", 50);
```

```
infix("?", 20, function led(left) {
    this.first = left;
    this.second = expression(0);
    advance(":");
    this.third = expression(0);
    this.arity = "ternary";
    return this;
});
```

```
function infixr(id, bp, led) {
    var s = symbol(id, bp);
    s.led = led || function (left) {
        this.first = left;
        this.second = expression(bp - 1);
        this.arity = "binary";
        return this;
    };
    return s;
}
```

```
function assignment(id) {
    return infixr(id, 10, function (left) {
        if (left.arity !== "name" &&
                left.id !== "." &&
                left.id !== "[") {
            left.error("Bad lvalue.");
        }
        this.first = left;
        this.second = expression(9);
        this.assignment = true;
        this.arity = "binary";
        return this;
    });
}
assignment("=");
assignment("+=");
assignment("-=");
```

```
function prefix(id, nud) {
    var s = symbol(id);
    s.nud = nud || function () {
        this.first = expression(80);
        this.arity = "unary"; };
    return s;
}

prefix("+");
prefix("-");
prefix("!");
prefix("typeof");
```

```
prefix("(", function () {
    var e = expression(0);
    advance(")");
    return e;
});
```

# Statement denotation

first null denotation

fud

```
function statement() {
    var exp, tok = token;
    if (tok.fud) {
        advance();
        return tok.fud();
    }
    exp = expression(0);
    if (!exp.assignment && exp.id !== "(") {
        exp.error("Bad expression statement.");
    }
    advance(";");
    return exp;
}
```

```
function statements() {
    var array = [];
    while (token.nud ||
           token.fud) {
        a.push(statement());
    }
    return array;
}
```

```
function block() {
    advance("{");
    var a = statements();
    advance("}");
    return a;
}
```

```
function stmt(id, f) {
    var s = symbol(id);
    s.fud = f;
    return s;
}
```

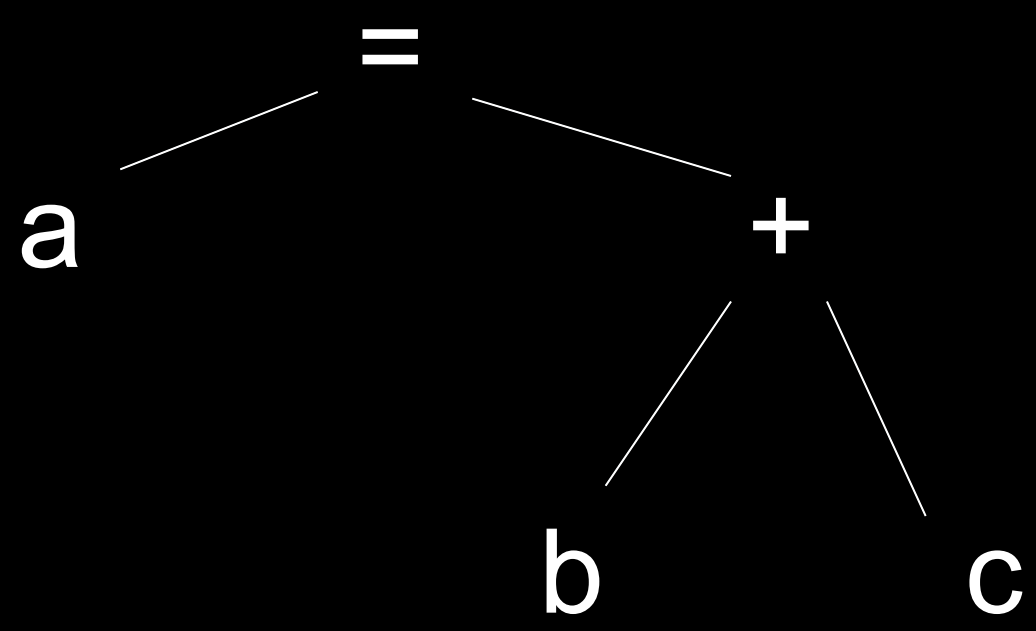
```
stmt("if", function () {
    advance("(");
    this.first = expression(0);
    advance(")");
    this.second = block();
    if (token.id === "else") {
        advance("else");
        this.third = token.id === "if"
            ? statement()
            : block();
    }
    this.arity = "statement";
    return this;
});
```

```
stmt("if", function () {
    this.first = expression(0);
    this.second = block();
    if token.id === "else" {
        advance("else");
        this.third = token.id === "if"
            ? statement()
            : block();
    }                                   // BCPL
    this.arity = "statement";
    return this;
});
```

```
stmt("if", function () {
    this.first = expression(0);
    advance("then");
    this.second = statements();
    if token.id === "else" then
        advance("else");
        this.third = statements();
    fi                          // Algol 68
    advance("fi");
    this.arity = "statement";
    return this;
});
```

```
function expression(rbp) {
    var left,
        tok = token;
    advance();
    left = tok.nud();
    while (rbp < token.lbp) {
        tok = token;
        advance();
        left = tok.led(left);
    }
    return left;
}
```

a = b + c;




Weave a stream of tokens into a tree

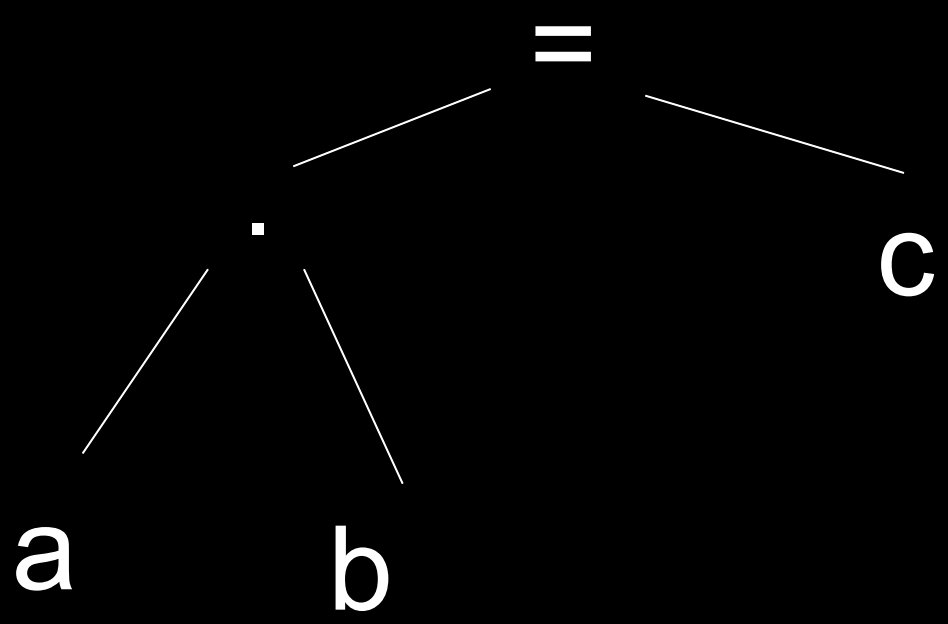
# a = b + c;

```
{
    id: "=",
    arity: "binary",
    first: {id: "a", arity: "word"},
    second: {
        id: "+",
        arity: "binary",
        first: {id: "b", arity: "word"},
        second: {id: "c", arity: "word"}
    }
}
```

# a = b + c;

```
statements()
    statement()
        expression(0)
            a.nud()
            while 0 < =.lbp
            =.led(a)
                expression(10)
                    b.nud()
                    while 10 < +.lbp
                    +.led(b)
                        expression(60)
                        c.nud()
```

a.b = c;




Weave a stream of tokens into a tree

# a.b = c;

```
{
    id: "=",
    arity: "binary",
    first: {
        id: ".",
        arity: "binary",
        first: {id: "a", arity: "word"},
        second: {id: "b", arity: "word"}
    },
    second: {id: "c", arity: "word"}
}
```

# a.b = c;

```
statements()
    statement()
        expression(0)
            a.nud()
            while 0 < ..lbp
            ..led(a)
                expression(90)
                    b.nud()
                    while 90 < =.lbp
            while 0 < =.lbp
            =.led(a.b)
                expression(60)
                c.nud()
```

# Top Down Operator Precedence

- It is easy to build parsers with it.
- It is really fast because it does almost nothing.
- It is fast enough to use as an interpreter.
- Dynamic: Build DSLs with it.
- Extensible languages.
- No more reserved words.

# Advice for language designers

# Minimalism

- Conceptual
- Notational

  Don't be cryptic
- Error resistant

  Confusion free
- Readable

  Can be easily and correctly understood by a reader

# Innovate

- We already have many Java-like languages.

```
CokeBottle cokeBottle = new CokeBottle();
```

- Select your features carefully.
- Beware of Sometimes Useful.
- Avoid universality.
- Manage complexity.
- Promote quality.

# Innovate

- Make new mistakes.
- Let the language teach you.
- Embrace Unicode.
- Leap forward.
- Forgotten treasure:
  State machines, constraint engines.
- Exploit parallelism.
- Distributed programming: clouds & cores.
- Have fun.

.

,

https://github.com/douglascrockford/TDOP

https://github.com/douglascrockford/JSLint

Beautiful Code: Leading Programmers Explain How They Think  [Chapter 9]
Oram & Wilson
O'Reilly

Thank you and good night.